Software Assignment

Manvik Muthyapu - AI24BTECH11021

Introduction

The main aim of this project is to write a code that should be able to compute eigenvalues of a matrix. To actually know what are eigenvalues, we need to think about linear transformations described by matrices. So, during any linear transformation, most vectors are knocked off their span (the line joining the origin to the tip of the vector), but there exists some special vectors which do stay on their span, so the effect of the linear transformation on these vectors is just to squish or stretch them. These special vectors of the transformation are termed as **eigenvectors** and the corresponding values by which they get stretched or squished are known as the **eigenvalues**.
Mathematical representation of an eigen vector is:

$$A\vec{v} = \lambda\vec{v}$$

where A is matrix representing the linear transformation and λ being the eigenvalue, and $\vec{v}$ is the eigenvector.
Now the above equation can rewritten as

$$A\vec{v} - (\lambda I)\vec{v} = 0$$

Then upon factoring out the eigen vector $\vec{v}$, we get

$$(A - \lambda I)\vec{v} = 0$$

Now, the possible solutions for above equation are the eigenvector to be zero or $det(A-\lambda I) = 0$. As we need non zero eigenvectors, we take $det(A - \lambda I) = 0$ as the solution. This is one of the basic and simplest of computing eigenvalues of a matrix.

Algorithm Selection

There are various algorithms to compute eigenvalues, depending on the properties of the matrix, numerical precision and stability, and size of the matrix. Some algorithms that I explored include:

- Characteristic Polynomial Method
- QR Algorithm
- Jacobi Method
- Divide and Conquer Method
- Lanczos Method

Any of these algorithms can be used to compute eigenvalues of a matrix. But each method has its own disadvantages.
Here are pros and cons for each of the algorithms stated above:

Characteristic Polynomial Method:

- Pros:
  - It can be applied for any matrix.
  - Through this method we can also find complex eigenvalues.

- Cons:
  - It becomes computationally very expensive for large matrices.
  - It gives less accurate answers and is numerically unstable for larger matrices.

QR Algorithm:

- Pros:
  - It is applicable for both symmetric and non-symmetric matrices.
  - This algorithm can compute both real and complex eigenvalues.
  - It gives pretty accurate eigenvalues and is numerically stable.
- Cons:
  - It can become computationally expensive for very large matrices.
  - It is not efficient for sparse matrices.

Jacobi Method:

- Pros:
  - It gives very accurate values in the case of symmetric matrices.
  - Implementation of this algorithm is simple.
  - It is numerically stable.
- Cons:
  - It is only applicable for symmetric matrices.
  - Convergence is slow.

Divide and Conquer Method:

- Pros:
  - It is efficient for symmetric matrices.
  - It is parallelizable, making it suitable for modern hardware.
- Cons:
  - It is only applicable to symmetric matrices.
  - Implementation is complex.

Lanczos Method:

- Pros:
  - It is efficient for large sparse symmetric matrices.
  - It is memory-efficient.
- Cons:
  - It is only applicable to symmetric matrices.
  - It does not directly compute all eigenvalues.

Out of all these algorithms, I think the best algorithm for general purpose is QR algorithm, as it can be applicable to both symmetric and non-symmetric matrices which most of the other methods cannot. It can also handle complex eigenvalue cases, so it is better than other algorithms.

QR Algorithm Description

The QR Algorithm as the name suggests is an iterative process that repeatedly decomposes a matrix into a product of two matrices, Q (an orthogonal matrix) and R (an upper triangular matrix):

$$A = QR$$

After decomposing, the next matrix in the iteration is given by:

$$A' = RQ$$

Repeat the process by taking A' as the new A. As the iterations progress, the matrix converges to an upper triangular matrix, whose diagonal elements will be the eigenvalues of the matrix taken originally taken (i.e A). A, A' and all other matrices along the process have the same eigenvalues. So the main idea here is using repeated QR decomposition, we are trying to transform the matrix into an upper triangular matrix because the eigenvalues for a triangular matrix are its diagonal elements.
Most important part in this algorithm is the **QR Decomposition**. There are various methods to do it. I have looked into three methods, they are:

- Gram-Schmidt Orthogonalization
- Householder Reflections
- Givens Rotations

Gram-Schmidt method even though is simple to implement, but only limited for small matrices, and it also numerically less stable. Givens Rotations is numerically stable, is efficient but it is best used for only sparse matrices. So I think the best method is Householder Reflections, because of its high numerical stability, efficiency, and also its applicability to dense matrices.

Householder Reflections

We have to compute $A = QR$, for that we zero out the elements below the diagonal of A to make it an upper triangular matrix. For this we use Householder matrices.
A Householder matrix H_k that zeros out elements below the diagonal in column k is given by:

$$H_k = I - 2vv^T$$

where $\vec{v} = \frac{\vec{u}}{\|\vec{u}\|}$ and $\vec{u} = \vec{r}_k - \begin{pmatrix} \|\vec{r}_k\| \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ where $\vec{r}_k$ is the column vector of the kth column in R from the diagonal element.
All of these Householder matrices are orthogonal. So when we multiply all these holder matrices we get,

$$H_n H_{n-1} \cdots H_1 A = R$$

from this we can get Q as $(H_n H_{n-1} \cdots H_1)^T = Q$.

Time Complexity

QR decomposition (Householder Reflections) requires $O(n^3)$ operations for an $n \times n$ matrix. And while applying the algorithm itself, a matrix converge generally after $O(n)$ iterations. So overall time complexity of QR algorithm for a general matrix is $O(n^4)$.

Observations and Comparisons

I have taken matrices of different dimensions and observed the eigenvalues and compared them with the values that are obtained when the same matrix is run in python using numpy package.

1. $$\begin{pmatrix} 3 & 5 & 4 \\ 8 & 3 & 5 \\ 4 & 3 & 1 \end{pmatrix}$$
   - Values given by numpy: 12.11357899 -3.32713327 -1.78644571
   - Values given by c-code: 12.113579 -3.327133 -1.786446
2. $$\begin{pmatrix} 3 & 5 & 4 & 2 & 1 \\ 8 & 3 & 5 & 1 & 7 \\ 4 & 3 & 1 & 9 & 2 \\ 6 & 8 & 2 & 7 & 4 \\ 5 & 9 & 2 & 3 & 6 \end{pmatrix}$$
   - Values given by numpy: 21.8143701 -4.64953030 0.0215161510+2.76421574i 0.0215161510-2.76421574i 2.79212794
   - Values given by c-code: 21.814370 -4.649530 0.021516+2.764216i 0.021516-2.764216i 2.792128
3. $$\begin{pmatrix} 3 & -7 & 5 & 9 & -2 & 1 & -3 \\ -7 & 1 & 6 & -9 & 4 & 3 & -2 \\ 5 & -3 & -3 & 2 & 8 & 4 & -1 \\ -3 & 8 & 2 & -1 & -4 & 1 & 6 \\ 8 & -1 & 5 & -1 & 2 & 5 & 9 \\ -6 & 3 & 1 & 8 & -2 & -8 & 1 \\ 5 & -3 & -6 & 2 & 1 & 7 & -1 \end{pmatrix}$$
   - Values given by numpy: -10.3344173 -8.9345089+5.45471038i -8.9345089-5.45471038i 6.18375327+10.05270822i 6.18375327-10.05270822i 4.41796428+0.81972239i 4.41796428-0.81972239i
   - Values given by c-code: -10.334417 -8.934509+5.454710i -8.934509-5.454710i 6.183753+10.052708i 6.183753-10.052708i 4.417964+0.819722i 4.417964-0.819722i
4. $$\begin{pmatrix} 4 & -1 & 9 & 2 & 6 & -3 & 7 & 0 & 5 & -8 \\ -3 & 5 & -2 & 8 & 1 & 6 & 0 & 7 & 9 & 4 \\ 6 & 0 & 4 & 7 & -5 & 9 & 1 & -3 & 2 & 8 \\ 2 & 8 & -3 & 5 & 7 & -4 & 6 & 9 & 0 & 1 \\ -6 & 7 & 0 & 4 & 3 & 1 & -2 & 8 & 5 & 9 \\ 5 & 1 & -7 & 2 & 6 & -3 & 4 & 9 & -1 & 8 \\ -2 & 5 & 9 & -1 & 4 & 8 & 0 & 6 & 7 & 3 \\ 8 & 6 & 7 & 1 & -4 & 5 & 3 & 9 & 2 & 0 \\ 4 & 3 & -1 & 9 & 2 & 6 & -3 & 8 & 5 & 7 \\ -5 & 9 & 2 & 7 & 3 & 4 & 8 & 0 & 1 & -6 \end{pmatrix}$$
   - Values given by numpy: 32.32313441 -7.12862011+14.5639155i -7.12862011-14.5639155i 10.16078498+7.10564715i 10.16078498-7.10564715i 4.86988331 -9.18721372 -3.48364051 -2.29324661+4.7159956i -2.29324661-4.7159956j
   - Values given by c-code: 32.323134 -7.128620+14.563915i -7.128620-14.563915i 10.160785+7.105647i 10.160785-7.105647i 4.869883 -9.187214 -3.483641 -2.293247+4.715996i -2.293247-4.715996i

5. $$\begin{matrix}
6 & -2 & 9 & 4 & 1 & 5 & 3 & 7 & 8 & -1 & 2 & -3 & 0 & 6 & 4 \\
1 & 4 & 5 & 3 & 6 & -2 & 7 & 9 & -1 & 8 & 2 & 3 & -5 & 4 & 0 \\
2 & 7 & 6 & -3 & 4 & 1 & 5 & 3 & 9 & 0 & 8 & 2 & -1 & 6 & -2 \\
5 & 9 & 3 & 7 & 0 & 8 & 2 & 1 & 6 & 4 & -2 & 5 & -3 & 7 & 9 \\
4 & 0 & 8 & 6 & 3 & 7 & 9 & 5 & 2 & -1 & 6 & 2 & 0 & 1 & 4 \\
7 & 5 & 2 & 3 & -1 & 6 & 4 & 9 & 1 & 8 & 0 & -2 & 7 & 3 & 5 \\
3 & 9 & 5 & 8 & 7 & 2 & 6 & 1 & -4 & 0 & 3 & 6 & -1 & 5 & 8 \\
9 & 1 & 0 & 5 & 3 & 7 & 4 & -2 & 2 & 8 & -1 & 9 & 6 & 3 & 7 \\
6 & 4 & 1 & 9 & 0 & 2 & 8 & 7 & -3 & 5 & 4 & 2 & 9 & -1 & 6 \\
0 & 8 & 7 & 4 & 9 & 5 & -2 & 1 & 6 & 3 & 9 & 0 & 7 & 4 & 2 \\
8 & 6 & 2 & 1 & 4 & 0 & 5 & 9 & -1 & 2 & 7 & 3 & 8 & -4 & 1 \\
3 & 7 & 4 & 5 & 6 & 9 & 2 & 8 & 0 & 1 & -3 & 9 & 2 & 7 & 5 \\
6 & 2 & 1 & 8 & 9 & 5 & 3 & 0 & 2 & 6 & -1 & 3 & 8 & 5 & 9 \\
1 & 8 & 9 & 4 & 5 & 7 & 6 & 0 & 3 & -2 & 7 & 1 & 9 & 4 & 6 \\
7 & 5 & 2 & 6 & 1 & 3 & 8 & 9 & 0 & 4 & 6 & 7 & 5 & 9 & 2
\end{matrix}$$

- Values given by numpy: 57.3314175 -5.83363956+11.16766987i -5.83363956-11.16766987i -8.55279531+4.93582869i -8.55279531-4.93582869i -7.54366576+1.81788784i -7.54366576-1.81788784i 10.01367098+8.73621035i 10.01367098-8.73621035i 10.76442274+2.74341167i 10.76442274-2.74341167i 2.96432854+7.15508472i 2.96432854-7.15508472i 3.49024315 4.55369607
- Values given by c-code: 57.331418 10.013671+8.736210i 10.013671-8.736210i -5.833640+11.167670i -5.833640-11.167670i 10.764423+2.743412i 10.764423-2.743412i -8.552795+4.935829i -8.552795-4.935829i -7.543499+1.818133i -7.543499-1.818133i 2.964162+7.154852i 2.964162-7.154852i 4.553696 3.490243